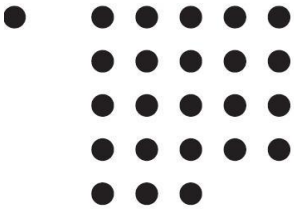


Fachhochschule Köln
Cologne University of Applied Sciences

Institut für Medien- und Phototechnik

Bachelorarbeit Medientechnik

Testautomatisierung von Online Spielen durch Objekt- und Bilderkennungsmethoden

Vorgelegt von

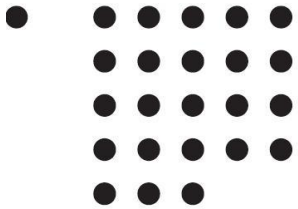
Alexander Schoenfeldt

Mat.-Nr. 11084259

Erstgutachter: Prof. Dr. Arnulph Fuhrmann (FH Köln)

Zweitgutachter: Robert Grzeskowiak (InnoGames GmbH)

Februar 2015



Fachhochschule Köln
Cologne University of Applied Sciences

Institut für Medien- und Phototechnik

Bachelor Thesis Imaging and Media Technology

Test Automation of Online Games Using Object and Image Recognition Approaches

Submitted by

Alexander Schoenfeldt

Mat.-Nr. 11084259

First Reviewer: Prof. Dr. Arnulph Fuhrmann (FH Köln)

Second Reviewer: Robert Grzeskowiak (InnoGames GmbH)

February 2015

Bachelorarbeit

Titel: Testautomatisierung von Online Spielen durch Objekt- und Bilderkennungsmethoden

Gutachter:

- Prof. Dr. Arnulph Fuhrmann (Fachhochschule Köln)
- Robert Grzeskowiak, M. Sc. (InnoGames GmbH)

Kurzfassung:

Moderne Browsergames sind immer komplexer werdende Programme mit sehr umfangreichem Quellcode. Mit zunehmender Komplexität sinkt die Wartbarkeit solcher Programme. Automatisierte Regressionstests vereinfachen den Wartungsprozess und ermöglichen Entwicklern und Testern ihre Arbeit effektiver zu gestalten.

Diese Bachelorarbeit thematisiert die Anwendung automatisierter Testverfahren auf Webapplikationen. Der Anwendungsfall wird an einem onlinebasierten Strategiespiel für den Browser vorgestellt. In der Einführung werden die Rahmenbedingungen für ein Softwareprojekt mit automatisierten Testverfahren dargestellt und verschiedene Ansätze der Testautomatisierung dargestellt. Im Hauptteil wird ein konkreter Lösungsansatz für Testautomatisierung, angewendet auf das Spiel „Forge of Empires“, vorgestellt. Das von der Firma erstellte Testframework wird vorgestellt. Dieses setzt sich aus verschiedenen Softwarekomponenten zusammen. Abschließend werden die präsentierten Methoden und Technologien vergleichend analysiert.

Stichwörter: Test Automatisierung, Online Spiele

Datum: 16. Februar 2015

Bachelors Thesis

Title: Test Automation of Online Games using Object and Image Recognition Approaches

Reviewers:

- Prof. Dr. Arnulph Fuhrmann (Fachhochschule Köln)
- Robert Grzeskowiak, M. Sc. (InnoGames GmbH)

Abstract:

State of the art browser games are increasingly complex pieces of software with extensive code basis. With increasing complexity, a software becomes harder to maintain. Automated regression testing can simplify these maintenance processes and thereby enable developers as well as testers to spend their workforce more efficiently.

This thesis addresses the utilization of automated tests in web applications. As a use case test automation is applied to an online-based strategy game for the browser. The introduction presents the general conditions for a software project that makes use of automated tests and different approaches of test automation. The main section introduces a specific solution of test automation applied on the game “Forge of Empires”, produced by Innogames GmbH. The created framework of the company is presented. It implements different software frameworks. The conclusion consists of a comparison and analysis about the presented methods and technologies.

Keywords: Test Automatisierung, Online Spiele

Date: 16. February 2015

Contents

1. Introduction 1

1.1 Automated Testing 1

1.2 Automated Testing at InnoGames 2

1.3 Department: Quality Assurance 2

1.4 Testing Patterns 3

2. The InnoAutomation Framework 5

2.1 Used Libraries 5

2.2 Class Structure 25

2.3 TestLink 28

2.4 Jenkins 28

2.5 Configuration 30

Refactoring: From Spaghetticode to PageObject Pattern 37

3. Conclusion 38

4. List of Literature 40

5. Eidesstattliche Erklärung 43

1. Introduction

1.1 Automated Testing

Testing is subject to the ongoing demand to achieve optimization and high quality in the software development process. The "BITKOM Guideline for Industrial Software Development" points out the following considerations when test automation is planned for a software project [TestAutm]. Besides the increasing complexity of software projects and the obligation to proof test results, test management has to work in compliance with the requirements of time- and cost targets. At the same time the test coverage ought to be as high as possible. But Test Automation is not able to completely exclude human failure, automated tests will not necessarily find more bugs than manual testing and the total costs do not have to be inevitably lower. Test Automation is considered to be profitable for long-term software projects with ongoing release cycles. The working hours saved by the repetitive retesting of codes enables testers to focus on new features to find new bugs while automated tests cover the already existing features that might be affected by code changes. The following section will examine reasons for integrating Test Automation in the software project.

Once the automated test cases are implemented they can be executed over and over the same way which terminates the possibility of creating an error by changing the test process. In addition, it is also only once necessary to create the test data which is needed for some tests. Automated tests ensure the reproducibility of errors via documentation to understand the root and nature of that respective error. The documentation created in combination with the constant execution of tests makes the software quality measurable once it has been defined. Furthermore it enables test engineers to compare test results after code changes, which would be too costly in most cases when retesting manually. Therefore, manual tests are often not retested once they succeeded, although, unpredictable side effects can produce new errors after the code has been changed. Test Automation makes it possible to do load- and performance tests in the first place. With manual tests it is most of the time not possible or not reasonable. The higher speed of automated tests compared to manual ones enables a significantly increased test coverage and reduces human error.

The paper "A Theoretical Review of the Impact of Test Automation on Test Effectiveness" outlines clearly "Unrealistic Expectation and gross Underestimation of Complexity" as the pitfalls of test automation [TAE]. Careful test planning and management is necessary and a thoroughly backup of the test process is needed, since it will not be possible to automate all of the test cases. While organizing the automation project, the manual testing should not be ignored or even be neglected. Furthermore, most cases require manual testing,

which is very costly in terms of time and additionally introduces human failure into the process.

Used correctly, Test Automation has the potential to significantly reduce the costs of testing, nevertheless, it should be considered that the results and its validity considerably depend on the test data, the test cases and the test coverage. These parts constitute the initial investment, because only the careful preparation of the automation project and the maintenance of test cases is leading to sustainable benefits.

1.2 Automated Testing at InnoGames

The InnoGames GmbH is a development company of browser- and mobile games located in Hamburg. The company has a strong focus on games from the strategy-genre and specializes in historical war settings. All games are set up based on the free-to-play business model. In all games players initially get full functionality of the game for free on all supported platforms without restrictions or time limits. The firms' business model is based on the option to buy premium accounts or items that offer additional advantages in the games. The company policy is to ensure an optimal game experience regardless of standard or premium account [InnoG].

Recognizing the opportunities of Test Automation InnoGames puts a lot of effort in developing a Test Automation Framework as the company is aware of the potential to reduce the workload needed in the game-testing process as well as the opportunity to improve the quality of future software developments.

1.3 Department: Quality Assurance

Since the change from matrix- to studio organization the Quality Assurance (QA) department of InnoGames operates more as a QA community. Every department or game studio has their own QA team, but these teams share their knowledge and discuss best practices in biweekly community meetings.

The task of every QA team is to support the ongoing development process in order to ensure the product quality and detect errors before they enter the final version. This is done by manual and automated testing of the specified requirements in the fields of graphic, performance, functionality and game logic. The goal is to maintain and increase the attention as well as the enthusiasm of the player.

The InnoAutomation Framework was utilized throughout the development of the Online Browser Games "Grepolis" and "Forge of Empires" (FoE). Both games

are strategic war games with historical background, which encourage the player to build an empire in order to fight against enemies and conquer the world.

The Grepolis Game uses HTML5 which makes it possible to automate test cases only using PageObjects. Forge of Empires instead is implemented with Adobe Flash which led to the decision to use image recognition software for test automation. Only the Registration or Landing Page tests are automated using PageObjects.

1.4 Testing Patterns

There are different possibilities to automate tests. The minimum quality criteria should be stability, maintainability and readability. The quality of technical implementation ensures the stability. The maintainability depends on the test workflow and the readability depends on the presentation layer. The following section introduces established patterns of test automation and discusses the advantages and disadvantages [Meth].

Capture & Replay

The creation of automated tests with capture & replay tools allows the QA tester to run the application and record the user interactions [C&R]. These recorded sessions can be automatically replayed without the need of a human user. Technical information for recognizing the interactive elements are not stored centrally but in separate files. This leads to high maintenance costs of GUI changes. Capture & replay tools are mainly used to record simple interaction sequences and not entire interactive sessions, because the lack of maintainability with increasing complexity of tests. The advantage of these tools is that QA testers without programming experience can use them.

Keyword-Driven Testing

Tests automated with this pattern are driven by keywords that represent actions of a test including input data and expected results [KDT]. With this pattern test cases can be written independently from the Software under Test (SUT). Once the SUT changes, the keywords must be refactored, but the test cases keep their validity. It also enables QA testers without prior experience to use automation tools that write and run test cases. The created tests are executed by a robot. The advantage of keyword-driven testing is the reuse of keywords in different test cases, which reduces the maintenance costs. Keyword-Driven testing can be used to create acceptance tests, because keywords can be created without implementation details. Due to the high effort

that is needed to create keywords with its functionality this pattern is only profitable for big or long term software projects.

Page Object Pattern

Tests created with the page object pattern are a compilation of test cases based on the GUI element. For instance a single web page can be represented as a page object [PGOP]. Every test case must access the GUI via the page objects. Page objects provide methods in order to access single elements of a web page. Advantages of this design pattern are the enhanced test maintenance and the reusability of the code. If the UI changes, tests do not need to be refactored, whereas only the page objects have to be [PageOb]. There is a clean separation between page specific code and test code. Services and operations provided by a page object are collected in one place and are not scattered in the code.

Model-Based Testing

While other patterns require the manual creation of test cases model-based testing is ought to create the test cases automatically based on a model [MBT]. Nevertheless the model has to be created manually. The model is created from the requirements of a software, mostly focused on functionality in order to simplify the model. The automated generation of test cases shall lead to a more efficient test process, because of the transparency and controllability of test creation. Furthermore this approach aims to have a person independent test quality.

2. The InnoAutomation Framework

In this section the InnoAutomation Framework will be introduced based on the FoE Automation Project.

The InnoAutomation Framework consists of different libraries, frameworks and self-written code. In the following sections the main components with its purposes and common functionalities will be presented to give a quick understanding of the project and its workflows in order to assure an efficient induction. Furthermore the class structure of the InnoAutomation Framework will be pointed out.

2.1 Used Libraries

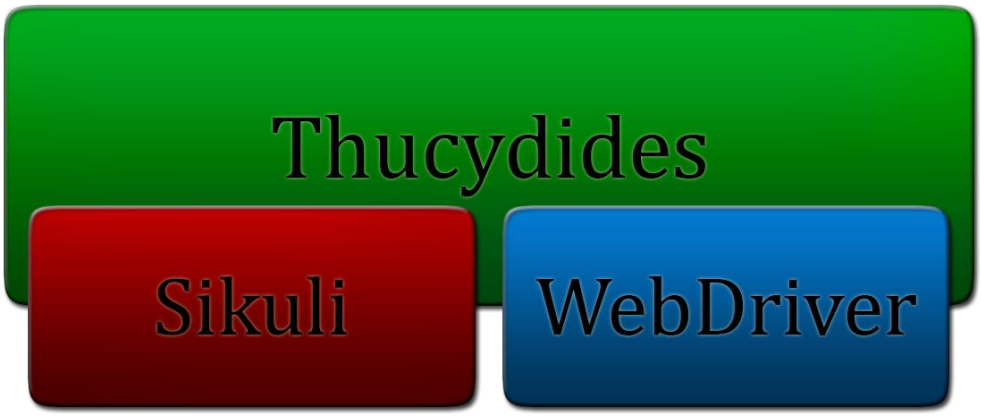


Illustration - 1 - Base of the InnoAutomation Framework

The basis of the framework is provided by libraries coming from mainly three Frameworks:

- Selenium 2 / WebDriver
- Thucydides
- Sikuli

Selenium 2 and Sikuli are operating on the same level while Thucydides is wrapping Selenium providing more advanced tools for writing tests easier.

The following sections give a more detailed insight in every of the above mentioned frameworks.

Selenium 2 / WebDriver

Selenium is a software testing framework that aims to automate the input of user interactions for web applications in the testing purposes[Sel]. With the release of Selenium 2 the WebDriver API has been included, which is an Object Oriented API that provides functionalities to drive a browser natively [WebDr].

Introduction

The InnoAutomation framework uses WebDriver to automate tests that simulate a realistic user interaction. When starting the test software, the simulated user opens the browser, types a URL, clicks on buttons, types fields and submits data. Furthermore automated tests with Selenium can check the visibility of elements in a web application [W3C].

How to use WebDriver

In the FoE Automation Project the WebDriver is implemented and declared in the SimpleTestTemplate.java class that extends all test classes.

```
@RunWith(ThucydidesRunner.class)
public abstract class SimpleTestTemplate {
    public final static String BASE_IMG_SRC = "src/test/resources/images/";
    protected final static int DEFAULT_SCREEN_NUMBER = 0;
    protected final static int DEFAULT_PAUSE = 5000;
    public final static String baserUrl = "http://yy.forgeofempires.com/";
    protected SikuliUtils sikuliActions;

    @Managed(uniqueSession = false)
    public WebDriver driver;
```

Code 1 – Initialization of WebDriver, SikuliUtils and constant values

That class manages what happens before and after every test execution and all available test steps (PlayerSteps, GuestSteps, [...]) are declared.

```
@ManagedPages
public Pages pages;

@Steps
public static LogsSteps logsSteps;

@Steps
public PlayerSteps player;

@Steps
public GuestSteps guest;
```

Code 2 - Initialization of Pages- and Steps-objects

The *setUp()* method takes care of business that has to be done right before every test like starting a specific browser (Chrome in this project, defined in thucydides.properties), opening the URL of the FoE website, maximizing the browser window, cleaning the test email account and initializing the *SikuliUtils sikuliActions*. The latter is used for image recognition and will be introduced later.

```
@Before
public void setUp() throws MalformedURLException, AWTException {
        int max = 1000;
        Robot robot = new Robot();
        robot.mouseMove((int) (Math.random()* max), (int) (Math.random()*
max));

        try {
            Runtime.getRuntime().exec(
"wscript src/test/resources/minimizeAll.vbs");
        } catch (IOException e) {
            System.exit(0);
        }
        driver.get(baserUrl);
        driver.manage().window().maximize();
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
        EmailChecker eMailChecker = new EmailChecker(true);
        eMailChecker.removeAllMessages();

        sikuliActions = new SikuliUtils(DEFAULT_SCREEN_NUMBER,
DEFAULT_PAUSE);
    }
```

Code 3 – The *setUp()* method is executed before every test

Once the test succeeded or failed the *tearDown()* method is applied. In case of a failure or unexpected event a screenshot that gets attached to the test report is created. Finally the browser is closed.

```
@After
public void tearDown() throws Exception {
        FailureDetectingStepListener failureDetectingStepListener =
new FailureDetectingStepListener();
        if(!failureDetectingStepListener.lastTestFailed()) {
            captureScreenshot(getClass().getSimpleName());
        }
        if(driver != null) {
            driver.quit();
        }
    }
```

Code 4 – The *tearDown()* method is executed after every test

To make interaction between WebDriver and web applications possible WebElements have to be defined in a way the WebDriver can interact with them [WebEl]. WebElements relate to website components, such as buttons, fields,

list elements, etc. After defining a WebElement user interactions can be simulated. In the FoE Automation the usage of this element is limited because the game itself runs with flash player, which does not consist of HTML elements. Nevertheless, all tests on the Webpage of FoE are handled with WebElements, which maintainability is higher than the image recognition approach used in the flash player.

Basic commands / API

WebDriver

The WebDriver class is an interface that represents an idealized web browser and provides the functionality to control the browser, selects WebElements and aids within the debugging process [WebDrITF].

The most important methods are *get(String)*, which loads the requested web page and the various methods to find WebElements like *findElement(by)*.

The following passage gives a quick overview to the common methods in the FoE project.

Common methods of WebDriver

void get(java.lang.String url)

Loads the passed web page in the configured browser. In case of the FoE-Project it is the Forge of Empires landing page.

The Parameter "*url*" is a fully qualified URL that is intended to load.

WebDriver.Options manage()

Provides the Option interface that delivers a lot more functionality to the WebDriver.

Use cases in the FoE Automation Project:

driver.manage().window().maximize();

- Maximizes the browser window

Explicit and Implicit Waits [ExImpl]:

Explicit: *Thread.sleep(long milis)*

- Waits for the given amount of time no matter if the condition is fulfilled or not.

Implicit: *driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);*

- Polls the DOM to wait the given amount of time if elements are not immediately available. Once the condition is fulfilled the next step continues.

WebElement

The WebElement is represented by a HTML element and performs commonly used interactions with a page [WebEl].

Calling a method of WebElement triggers a check that validates the reference to the element.

Common methods of WebElement

void click()

The *click()* method performs a mouse click on the referred element. If a mouse click causes the loading of a new page the methods blocks further steps until the page has loaded. An element only can be clicked when it is visible.

The FoE Automation project uses that method to click login buttons or to choose the world which should be played in.

void sendKeys(Charsequence… keysToSend)

This method simulates typing a *String* into an element that is able to get such input.

Used in the FoE Automation project to type a user name and email address when registering a player and logging into the game with the credentials.

boolean isDisplayed()

Returns *"true"* when the element is displayed and *"false"* when it is not. Can be used to assert tests.

Thucydides

In this section the Thucydides library will be introduced and classed in the InnoAutomation Framework [Thucyd]. In addition a description of the implicit reporting system and the integration in Jenkins as well as the functionality given in the FoE Automation project follows.

Introduction

The Thucydides open source library provides a set of tools to write Selenium 2 tests easier [ThucydIntro]. It sends reports to Jenkins offering detailed and narrative test reports. It also enables the evaluation of all tests together for a high level feedback that documents the projects progress and status.

The detailed information can be used by the software developers to test and update the code while the high level views and reports give a good overview to product managers and team leads.

This is done by turning the automated web tests into automated acceptance criteria as designated in Acceptance Test Driven Development (ATDD).

How to use Thucydides

There are different places in the code of the FoE Automation project where Thucydides comes to use.

The in the WebDriver section introduced SimpleTestTemplate.java implements some annotations of Thucydides which will be focused on in the following.

SimpleTestTemplate.java

The only things to find in this class by Thucydides are the annotations that manage the test.

@RunWith

This annotation shows that this class is a test run by Thucydides. The ThucydidesRunner.class [ThucydAnnot] “initializes a WebDriver instance before running the tests in their order of appearance. At the end of the tests, it closes and quits the WebDriver instance.” [ThucydRun] During the tests the runner provides a StepFactory which invokes the test steps and notifies the runner about the step outcomes. Furthermore it processes any Thucydides annotation in the test classes and provides a reporter that is able to report in XML and HTML but also can be extended by subscribing more reporter implementations.

```
@RunWith(ThucydidesRunner.class)
public abstract class SimpleTestTemplate {
```

Code 5 – The @RunWith annotation lets Thucydides initialize a WebDriver instance

@Managed

The @Managed annotation of the public WebDriver field allows Thucydides to open, close and use the WebDriver in the test pages and test steps.

The parameter uniqueSession is set to false which means that the browser gets restarted after every test to ensure that each test is independent. The Thucydides runner instantiates this WebDriver with the current WebDriver.

```
@Managed(uniqueSession = false)
public WebDriver driver;
```

Code 6 – The @Managed annotation enables Thucydides to access the WebDriver instance

@ManagedPages

In order to use the PageObject Pattern the Pages class annotated with this annotation is provided with instantiated PageObjects by the Thucydides runner.

```
@ManagedPages
public Pages pages;
```

Code 7 – The Thucydides runner provides instantiated PageObjects for Pages annotated with @ManagedPages

@Steps

This annotation defines the class that has all steps related to the user. “For high-level acceptance or regression tests [RegTest], it is a good habit to define the high-level test as a sequence of high-level steps. It will make your tests more readable and easier to maintain if you delegate the implementation details of your test (the "how") to reusable "step" methods.“ [ThucydAnnot] *PlayerSteps* are all steps related to a user that is logged into the game. *GuestSteps* define all steps related to the user that is not already logged in.

```
@Steps
public PlayerSteps player;

@Steps
public GuestSteps guest;
```

Code 8 – Objects with the @Step annotation define different step types.

Pages

The FoE Automation project uses Page-Object-Pattern, which will be explained in the PageObject section in detail. Every class that is extended by the class PageObject is a different webpage or layer on the FoE landing page and has different WebElements. The distinction has to be clear, because some elements might not be available anymore if a new page or part of a page loads, which could lead to unwanted reactions [PageOb].

To test the login and registration process with WebDriver there are currently three pages necessary:

- LandingPage: http://yy.forgeofempires.com/ - The website where a potential user gets redirected to register to the game
- LoginPage: http://yy.forgeofempires.com/page/ - When the user has logged in from the landing page he is on the login page where he can click the play- or logout-button.
- WorldSelection: Is a new layer of the login page but has the same URL. Here it is possible to choose a world to play in.

Each of these pages contain WebElements, which are wrapped by WebElementFacade – a wrapper class from Thucydides providing tools for WebDriver. To find the web elements on the website Thucydides uses the @FindBy[1] annotation from WebDriver [WebDrAn]:

@FindBy

The @FindBy [FindByAnnot] annotation is placed above a WebElementFacade on a specific PageObject. In *Code 9* below you can see that with this annotation the name field to register a user is located via XPath. There are in addition several other methods to locate web elements. Here are some examples to access HTML elements via different ways using the @FindBy annotation.

@FindBy(xpath = "//input[@id='login_password']")

@FindBy(id = "login_password")

@FindBy(css = "div#form_element_password>label.register_password")

```
public class LandingPage extends PageObject {
    @FindBy(xpath = "//input[@name='register_name']")
    WebElementFacade registerNameField;
```

Code 9 – The *LandingPage* class owns all HTML elements related to the Landing Page website

This reference makes the WebElementFacade elements unique and ready for use. Due to the fact that they are private they have to be accessed via methods. The only useful action regarding the *registerNameField* is to type text in there. So a method called "*sendKeysToRegisterNameField*(String name) {}" is created. The naming [ThucydIntro] of the method and the element are also important parts of the Page-Object-Pattern [POP], because it ensures an understandable workflow of the presentation layer.

```
public void sendKeysToRegisterNameField(String name) {
	registerNameField.sendKeys(name);
}
```

Code 10 – A method to write text into the name field of the register section on the landing page

Tests

Every test is an extension of the SimpleTestTemplate.java and has the @WithTagValuesOf annotation of the Thucydides library.

@WithTagValuesOf

This annotation makes it possible to add tags to every test in order to classify them into different types that can be evaluated afterwards by the reporting system of Thucydides which will be explained in the section "Reporting System" [ThucydReport].

```
@WithTagValuesOf({"feature:army management", "story:basic
functionality", "level:battle", "top-level:smoke"})
public class FOE448 extends SimpleTestTemplate {
```

Code 11 – Tags are fetched by Thucydides and can be used as a filter in order to evaluate test results

Existing tags in the FoE Automation project are:

- Feature:
  - Settings
  - Army Management
  - Buff System
- Story – Is a part of the feature:
  - Collect all
  - Motivate all
  - Idle
- Level - Determines the test suite in which the test is running:
  - Basic

    - Tutorial
    - Buff System
- Top Level – Indicates the test category
    - Smoke
    - Master

Reporting System

Thucydides provides the FoE Automation project with a comprehensive reporting system which gives high level [ThucydHLRep] and low level reports. High level reports show diagrams and statistics to give an overview about the project status. Low level reports show the process of every step in a test and makes a screenshot when a test fails. This section will illustrate how Thucydides reports in the FoE Automation project.

Local Tests

Thucydides gives two options to check test results locally.

Console Test Reports

Within the console of the IDE Thucydides gives a quick overview in the console during test execution which writes line by line the executed step. The advantage of this method is to directly see which method fails. Since almost every method includes only one action it is relatively easy to find the source of the error.

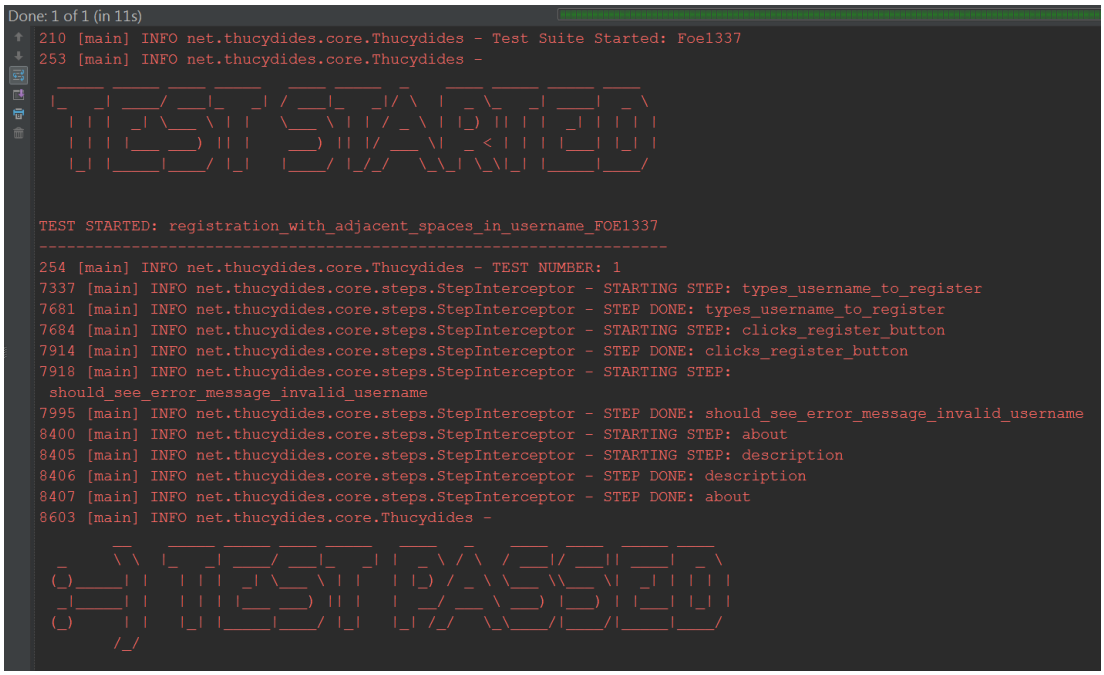


Illustration - 2 - Local console output of Thucydides after text execution

Once the test has passed the confirmation message will be displayed.

HTML Test Reports

The same report will be saved locally on the running computer in the \...\target\site\thucydides folder of the test automation project. The basic difference here is that the duration of every test step is illustrated, which can be relevant for performance evaluation in a later stage. In addition a screenshot of the last step or the step that failed is attached.

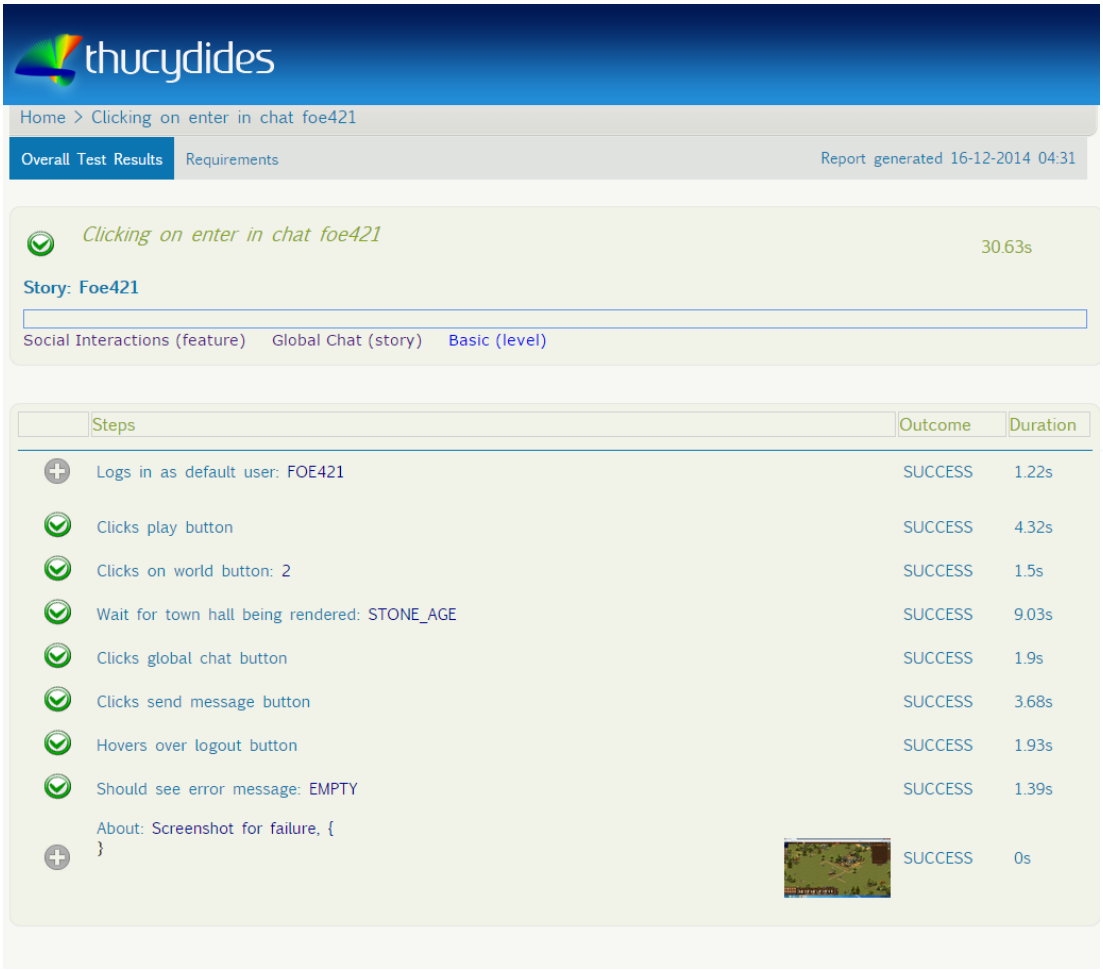


Illustration - 3 - Local Thucydides HTML report

Jenkins

In the FoE Automation project Jenkins is used to manage and automate the execution of tests on servers. The integrated Thucydides plugin [ThucydPlgnJen] provides also statistics that give an overview about the status of tests.

The illustration below shows the reports from Thucydides in the Jenkins plugin. A pie-chart gives a clear visualization of how many tests passed, are pending, were ignored, have failed or contain errors. This gives a fast feedback about the general status of the test cycle. Further information are provided in the test result summary where one can see the same information from the pie-chart in numbers. Below that the evaluation of the tags take place, which are defined in every test with the *@withTagValuesOf* annotation. In this high level view of the given example one can see that tests are failing related to Social Interaction features. Looking into the Stories tag shows more precisely where failures occur. In this scenario errors are related to the “Neighbor Chat” user story [UserStory]. Then in the section below it is possible to see explicitly which test failed. This evaluation of tests makes it very fast and comfortable to track the error down to its source.

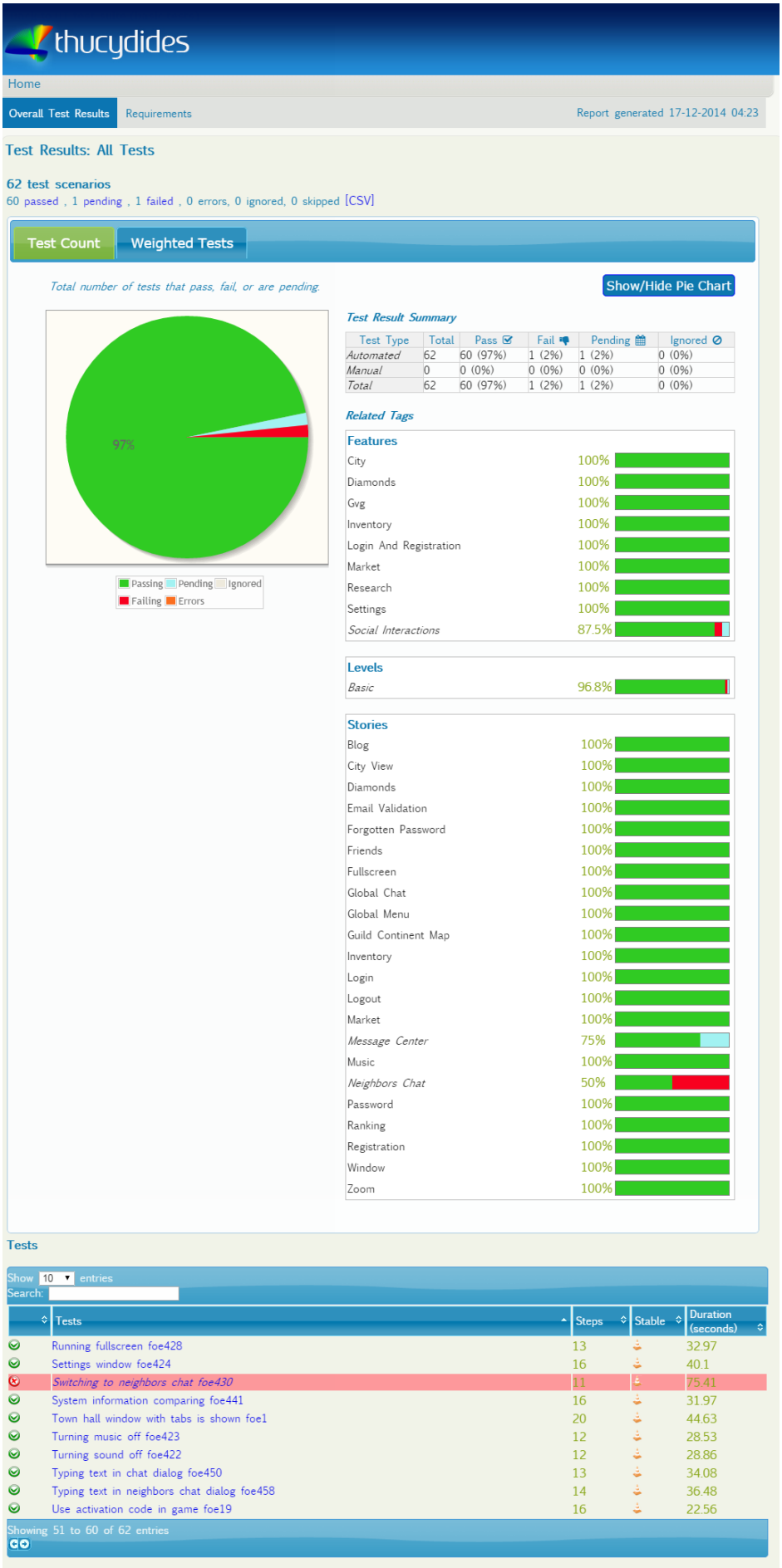


Illustration - 4 - Thucydides report in Jenkins

Prospects

Currently the existing FoE Automation framework is used to test the game after new features have been implemented, which does not involve the QA in the development process.

“Testing an application after it has been developed has a number of significant drawbacks. Most importantly, having feedback about problems raised at this late stage of development makes it very difficult to correct bugs of any size. This results in costly rework, wasted developer time, and delayed deliveries.” [ATDD]

The FoE Automation framework addresses the above described issue by turning the process of game development into ATDD, which basically means that the creating of automated acceptance tests begins before the development process starts. This kind of development process includes not only the QA team but the whole team by creating acceptance criteria. This process adjustment does not only increase the efficiency of testing but also ensures that every team member knows which feature will be implemented. Once the acceptance tests are written Thucydides provides the whole team with a comprehensive progress bar.

“Acceptance tests are also the ultimate progress indicators. An automated acceptance test works or it doesn't -- there is no "80 percent done" in ATDD!” [ATDD]

Product Managers will benefit from the high level reports giving a good indicator of the project status. They are given indication which feature is implemented and which is not, while developers have a clear overview of what has to be done from a user perspective.

Besides all these benefits given, the project will receive a broad set of regression tests, that have a great readability and maintainability, once the features are implemented.

Basic commands/API

Annotations

The annotations are documented in the “How to use Thucydides” section.

Methods

WebElementFacade click()

- Clicks on an element once it is visible and enabled

boolean isCurrentlyVisible()

- Returns true/false whether the element is visible on the screen or not.

WebElementFacade waitUntilVisible()

- Holds the test until the element is visible.

WebelementFacade

WebElement

The WebElement is documented in Selenium 2 API

Wrapselement

Is an interface illustrating that this class is wrapping an element.

WebElement getWrappedElement()

- Returns the wrapped WebElement

Webelementstate

This interface presents the state information about a WebElement.

Page Object Pattern

The PageObject Pattern is a design pattern where each screen of a web app is a sequence of objects and encapsulated features, which means that on every object or HTML element different methods can be executed. This automatically reduces the amount of duplicated code due to reuse in different test cases.

“A page object is an object-oriented class that serves as an interface to a page of your AUT” [POP] (Application under test).

The PageObject class provided by Thucydides represents the screens of a web page as a series of objects. Every class that is extended by the PageObject class is used to store WebElements wrapped by the wrapper class WebElementFacade as already mentioned in the section “Thucydides/ How to use Thucydides/ Pages” (Page 10).

The PageObjects represent the lowest level of the Page-Object-Pattern and are the place where the HTML elements get defined. The well-structured PageObjects and levels of abstraction above the PageObjects are used to increase the maintainability [ATDD] of the project. For example, if some UI element on the web page changes, only that specific PageObject has to be fixed and nothing else in the code. All other classes are not affected.

The following section demonstrates how tests with this type of pattern are written.

Tests with PageObjects

As mentioned above it is possible to reuse code as it is required in different tests. The following example will serve to show how it is done in a real test of the FoE Automation project.

```
@WithTagValuesOf({"feature:army management", "story:basic functionality",
"level:battle", "top-level:smoke"})
public class FOE447 extends SimpleTestTemplate{

@Test
public void army_management_era_filters_all_ages_FOE447() throws
IOException,InterruptedException,TimeoutException{


guest.atLandingPage().logs_in_as_default_user(getClass().getSimpleName());
player.atLoginPage().clicks_play_button();
player.atWorldSelection().clicks_on_world_button(2);

player.atCityScreen().waits_for_town_hall_to_be_rendered(ScenarioStepsTempla
te.Eras.IRON_AGE);

player.atGameHud().clicks_army_management_button_and_checks_for_opened_windo
w();

//Checking for bronze age
player.atArmyManagementWindow().clicks_era_selection_arrow();
player.atArmyManagementWindow().hovers_over_close_button();
```

```
player.atArmyManagementWindow().clicks_to_choose_era_in_era_selection(Scenar
ioStepsTemplate.Eras.BRONZE_AGE);
player.atArmyManagementWindow().should_see_units_in_unit_pool(AtArmyManageme
ntWindow.UnitPool.BRONZE_AGE_CHAMPION_ROUGE);

//Checking for All Ages
player.atArmyManagementWindow().clicks_era_selection_arrow();
player.atArmyManagementWindow().hovers_over_close_button();
player.atArmyManagementWindow().clicks_to_choose_era_in_era_selection(Scenar
ioStepsTemplate.Eras.ALL_AGES);
player.atArmyManagementWindow().should_see_units_in_unit_pool(AtArmyManageme
ntWindow.UnitPool.ALL_AGES_UNITS);
```

Code 12 – Example of an automated test of the game Forge of Empires

Every test is a sequence of method calls. The naming in combination with the PageObject Pattern makes it very easy to understand what happens in the test and it is done step by step just by reading the method call. This high level implementation does not question the technical details about the feature and hides them in the PageObjects. Once the steps are defined the tests can be written easily through reuse of the steps, which is occurring very often in game testing. Each class consists of only one test (@test) which is named like the test it performs plus a consecutive number that is also the class name.

Sikuli

Sikuli [Sik] is the image recognition tool that is used to automate the test steps within the game as it is not possible to use PageObjects. The basic concept of this method is to compare pre saved pictures from the tests with pixels on the screen. If Sikuli finds a match it performs the requested action, which is in most cases a mouse click.

Introduction

Once the user logged into the game in order to play Adobe Flash takes over. Since there are no HTML elements the WebDriver cannot work here. The SikuliUtils class provides the commonly used Sikuli methods and is implemented in the ScenarioStepsTemplate class which extends every test class. The following section will show how Sikuli is used for automated testing in the project.

How to use Sikuli

Since for guest steps WebDriver is used, all Sikuli actions are implemented in the player step classes,. Further all player step classes are extended by the ScenarioStepsTemplate class that declares the *SikuliUtils* object *sikuliActions*.

The *AtArmyManagementWindow* class will serve as an example. The army management is the window in the game where you can choose units to defend your city or attack other players or NPC's (Non Player Characters).

The *@Step* annotation imported from the Thucydides library is on top of every method indicating that this method has to be handled and later reported as a step in the evaluation. The first method is named based on the way a user does the interaction. In this case the name is *clicks_close_button_and_checks_for_closed_-army_management_window*. As one can see this is exactly what happens. First the Sikuli driver clicks on the close button and then an assertion is made by checking if the army management page header has vanished by calling the *sikuliActions().isImageNotPresent*(…) method. All methods that are implemented in the different step classes can be reused in every test.

```
public class AtArmyManagementWindow extends ScenarioStepsTemplate {

    @Step
    public void
clicks_close_button_and_checks_for_closed_army_management_window() throws
IOException, TimeoutException, InterruptedException {
        sikuliActions().clickOnImage(Commons.CLOSE_X_BUTTON);
        Assert.assertTrue("Army Management window not closed",
sikuliActions().isImageNotPresent(ArmyManagement.PAGE_HEADER));
    }

    @Step
    public void clicks_ok_button_and_checks_for_closed_window() throws
IOException, TimeoutException, InterruptedException {
        sikuliActions().clickOnImage(Commons.Ok_BUTTON_ORANGE);
        Assert.assertTrue("Army Management window not closed",
sikuliActions().isImageNotPresent(ArmyManagement.PAGE_HEADER));
    }

}
```

Code 13 – Example of Step methods

Basic commands/API

The SikuliUtils class is provided by the InnoAutomation Framework and provides methods that gather common used Sikuli methods and objects.

SikuliUtils.java

void clickOnImage(String)

Clicks on the center of the image that is passed with the *"String"*, which has to be the path to the image. Within that method a *"Mouse"* object is created and instantiated which performs the *".click()"* method.

```
public void clickOnImage(String imageName) throws IOException
    {
        // Click the center of the found target
        Mouse mouse = new DesktopMouse();
        mouse.click(getImage(imageName).getCenter());
    }
```

Code 14 – The *clickOnImage()* method is used to simulate a mouse click

void doubleClickOnImage(String)

Double clicks on the image that is passed with the *"String"*, which has to be the path to the image. Within that method a *"Mouse"* object is created and instantiated which performs the *".doubleClick()"* method.

```
public void doubleClickOnImage(String imageName)
            throws IOException
    {
        // double click the center of the found target
        Mouse mouse = new DesktopMouse();
        mouse.doubleClick(getImage(imageName).getCenter());
    }
```

Code 15 – The double*ClickOnImage()* method is used to simulate two fast mouse clicks

void clickOnSideOfImage(String, SideOfImage)

Clicks on the side of the image that is passed with the *"String"*, which has to be the path to the image. SideOfImage is an enumeration and enables Sikuli to click in every corner of an image. Within that method a *"Mouse"* object is created. The *SideOfImage* enumeration gets passed to a *"switch-case"* statement where it gets evaluated.

```
public void clickOnSideOfImage(String imageName, SideOfImage sideOfImage)
throws IOException
    {
        Mouse mouse = new DesktopMouse();
        switch(sideOfImage) {
            case CENTER: mouse.click(getImage(imageName).getCenter());
                break;
            case LOWER_LEFT_CORNER:
mouse.click(getImage(imageName).getLowerLeftCorner());
```

```
                break;
            case LOWER_RIGHT_CORNER:
mouse.click(getImage(imageName).getLowerRightCorner());
                break;
            case UPPER_LEFT_CORNER:
mouse.click(getImage(imageName).getUpperLeftCorner());
                break;
            case UPPER_RIGHT_CORNER:
mouse.click(getImage(imageName).getUpperRightCorner());
                break;
        }

    }
```

Code 16 – The *clickOnSideOfImage()* method is used to click in a specific region of an image

void hoverMouseOver(String)

Hovers over the image that is passed with the *"String"*, which has to be the path to the image. Within that method a *"Mouse"* object is created and instantiated which performs the *".hover()"* method on the center of the image.

```
public void hoverMouseOver(String imageName) throws IOException {
        Mouse mouse = new DesktopMouse();
        mouse.hover(getImage(imageName).getCenter());
    }
```

Code 17 – The *hoverMouseOver()* method is used to hover over an image

boolean imageIsPresent(String)

Checks if the image is present on the screen. Returns *"true"* or *"false"*. A ScreenRegion object from Sikuli calls the *"getImage()"* method where the image of the passed String source is compared with the current screen. Once there is a match the method returns true or after five seconds without finding a match it returns false.

```
public boolean isImagePresent(String imageName) throws IOException
    {
        ScreenRegion region = getImage(imageName);
        return region != null;
    }
```

Code 18 – The *imageIsPresent()* method asserts if the searched image is currently displayed

2.2 Class Structure

This section will give a look into the folder structure of the project and document its content and purposes for the automation.

The following illustration shows the folder structure of the FoE Automation Project.

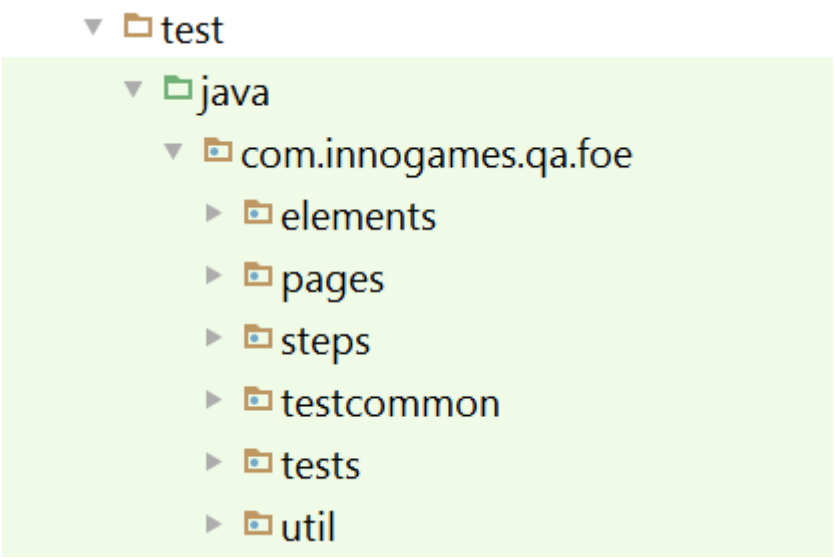


Illustration - 5 - Folder Structure of the InnoAutomation Framework

com.innogames.qa.foe

elements

The elements folder consists of different classes that represent specific parts of the game. Within these classes are Strings that hold a path of an image resource. Sikuli accesses these Strings to compare the images with the pixels on the screen. If the pixels match the test step succeeds. Putting these image resources in objects enables code reuse and clears the code.

pages

The pages folder consists of different classes that represent the FoE landing page or parts of it. Within these classes are the already introduced WebElementFacade objects and methods to interact with them.

steps

The steps folder consists of the guest, the player folder and the ScenarioStepsTemplate class that hand down the common functionalities for the player- and guest step classes. Furthermore there are the GuestSteps and

PlayerSteps class which provide getter methods for the classes that are in the guest- and player folder.

guest
The guest folder consists of different classes that provide the reusable steps (@Step) for the tests in the tests folder, which are executed as a guest e.g. logging in on the landing page.

player
The player folder consist of different classes that provide the reusable steps (@Step) for the tests in the test folder, which are executed as a player e.g. clicking the army management button.

testcommon
The testcommon folder consist of different classes that serve as a template for all test classes in the tests folder providing basic functionality for every test as for instance starting and closing the browser or logging into the game.

tests
The tests folder consists of several subfolders that organizes the contained test classes according to their feature. For example all tests related to the battle are in the battle folder which also has subfolders that specifies the features functionalities such as army management and continent.

util
The util folder consists of several classes that are useful helper and can be extended on demand.

ClipboardUtil
Provides methods to interact with the system clipboard.

EmailChecker
Provides methods to interact with the email account that is used for registering the accounts like logging in, removing all messages, getting the latest email, etc.

RandomStringGenerator
Provides a method to generate a random String.

WindowTools
Provides methods to interact with the browser window as for instance switching tabs or windows, handling the window size, etc.

Annotations

The InnoAutomation Framework uses JUnit annotations for the test template SimpleTestTemplate.java

@BeforeClass

Methods with the @BeforeClass [BeforeClass] annotation are executed right before every test sequence. This can save the computation of expensive setups but might compromise the independence of tests.

@Before

Methods with the @Before [Before] annotation are executed right before every test. This annotation can be used to avoid code repetitions in every test. In the InnoAutomation projects it is mainly used to start the browser, maximize the browser window and clear the test email account.

@Test

Methods with the @Test [Test] annotation are run as a test. If exceptions are thrown Thucydides will report a failure within the test.

@After

Methods with the @After [After] annotation are executed right after every test. This feature is used for taking screenshots in case of failing tests and shutting down the browser when finishing a test.

@AfterClass

Methods with the @AfterClass [AfterClass] annotation are executed right after every test sequence. This can save the computation of expensive setups but might compromise the independence of tests.

Example:

This illustration demonstrates how the workflow of these annotations.

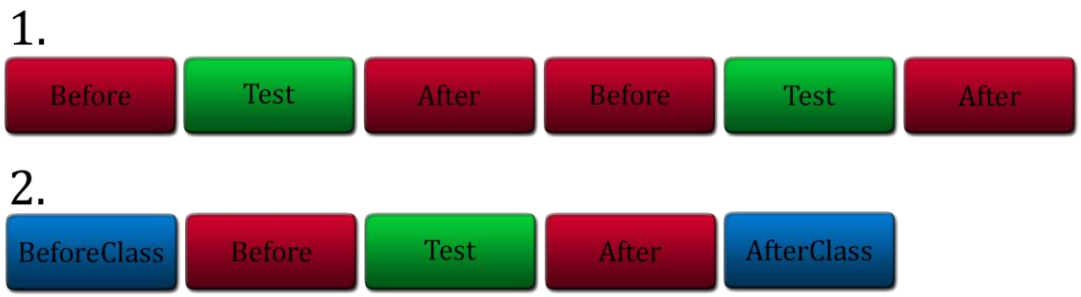


Illustration - 6 - Workflow of test execution using JUnit Annotations

@Managed(uniqueSession = true|false)

@Managed lets Thucydides take care of the following WebDriver. "uniqueSession" decides whether the browser should be restarted after every test case or if all tests shall be run in the same browser.

ScenarioSteps

The ScenarioSteps class provides a set of reusable steps for web tests. For example it implements the *getDriver()* method that returns the WebDriver from the pages that are used.

2.3 TestLink

"TestLink is a web-based test management system that facilitates software quality assurance." [TestLink] The FoE Automation Project uses it to organize tests. A plugin enables Jenkins to collaborate with TestLink in order to exchange the status of tests. Every test case is registered in TestLink and is ordered in the same structure as in the automation project itself. This enables the team to have a proper overview about the test coverage.

2.4 Jenkins

"Jenkins is an open source continuous integration tool written in Java." [Jenkins] The QA of InnoGames uses Jenkins to monitor the execution of repeated jobs like the nightly execution of the automated test cases. A plugin integrates Git into Jenkins and allows to manage the source codes including merging of code and building of new versions.

In the illustration below shows the Build Pipeline used in the FoE Browser Automation Project. Every row is an apposition of different jobs that need to be done in order to execute the tests. The first job (TestRestoreWorld) is to restore the test accounts on the server to their initially prepared state to ensure the independency of the test. The next job (Smoke Test) is a selection of test cases that are supposed to ensure the basic functionality of the test. The feedback is much faster than the last job (Master List) which makes it possible to react quicker to errors and bugs. The Master List covers all parts of the game and tests its functionality on a very detailed level. This regression test checks if parts of the game are affected by code changes or merges.

Illustration - 7 - The visualization of the automated browser test automation workflow in Jenkins

2.5 Configuration

Slave Server

InnoGames uses Jenkins to manage the servers which execute the test cases, which makes it possible to access the server and execute the tests anywhere. All that is needed is a Notebook with a VPN connection and access to Jenkins.

Standalone machine

The standalone machine is a mac mini and can be accessed via an URL in a web browser that redirects the user to the Jenkins login page. Once logged in it is possible to run Jenkins jobs as described in the “Jenkins” section.

Cloud Service

The Grepolis Automation Team uses Browserstack [browserStack], a cloud service, which makes it possible to use all available browsers with different versions and operating systems, for testing the project.

Tutorial – How to write a test case

This section will use an example to illustrate how a test in the FoE Automation project is conducted.

The generic test case that will be automated has the purpose to check if an error message will appear during registration process when the user types in a user name with adjacent spaces like "A B".

All automated and manual tests are listed in the web-based test management system Testlink

Creating a test in TestLink

Follow the subsequent steps to create a test case in TestLink:

1. Login to https://testlink.innogames.de/login.php with your account.
2. Choose "FoE" as the Test Project

Illustration - 8 - Choosing the project in TestLink

3. Click on Test Specification in the Test Specification bar on the left.

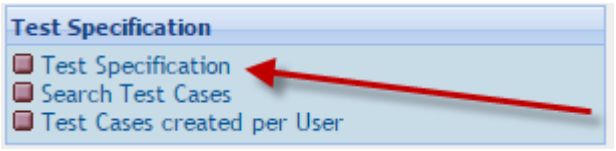


Illustration - 9 - Clicking the Test Specification to access all tests in TestLink

4. Choose a fitting Folder for the test case in order to create it there. For the given example the Folder "Login & Registration" is a fitting section.

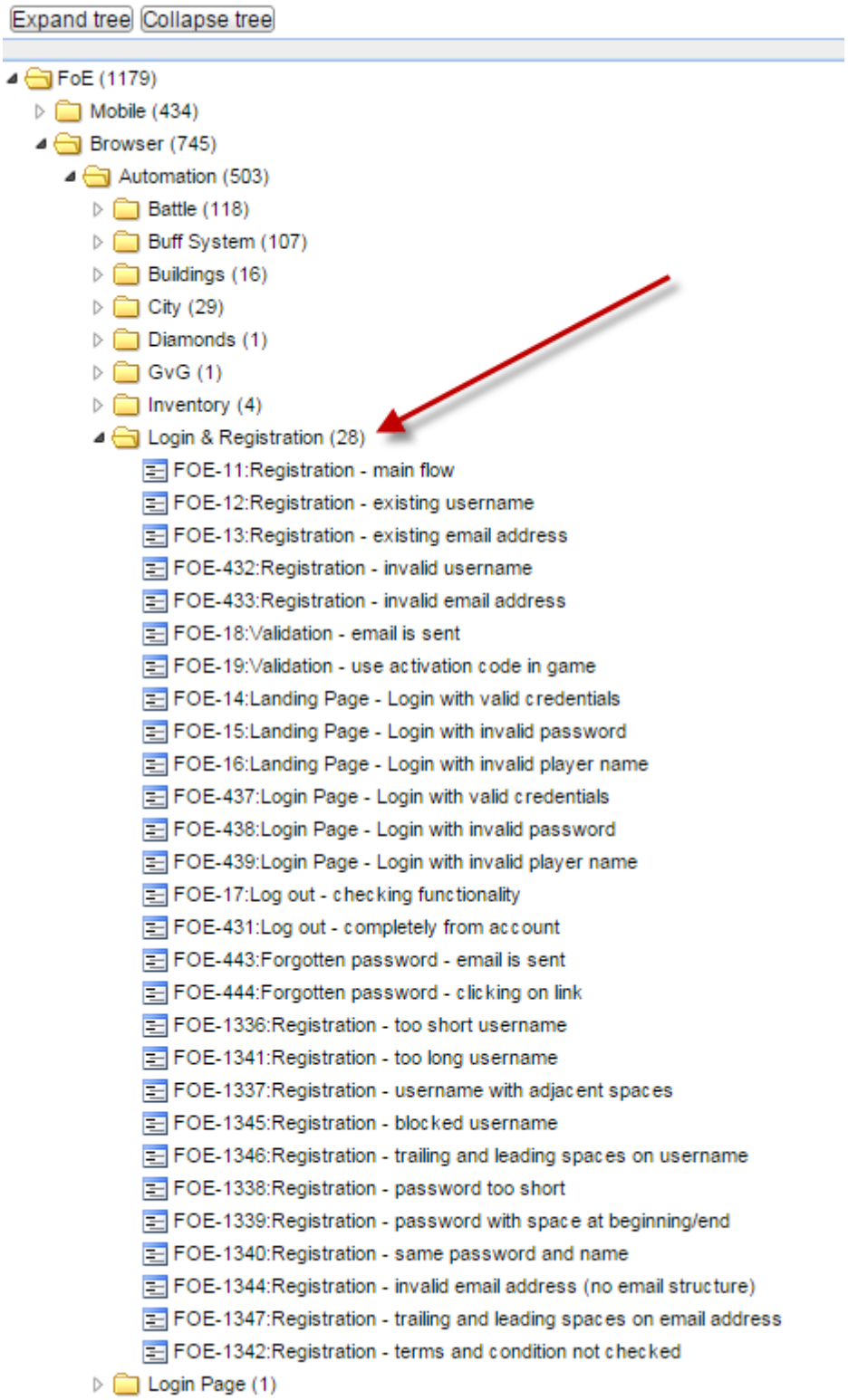


Illustration - 10 - Choosing an accurate folder to create the test

5. Click on the small gear icon in the main section to expand the possible test case operations.

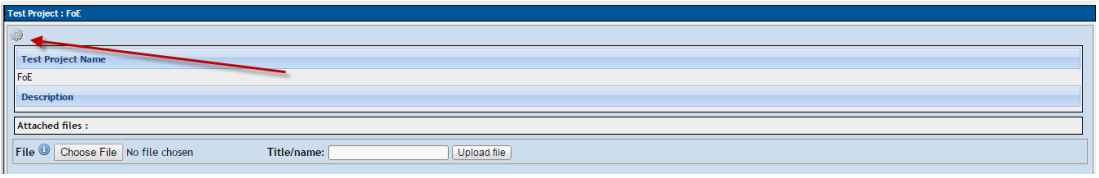


Illustration - 11 - Expanding possible test case operations

6. Click on the "Create" Button to create a test case.

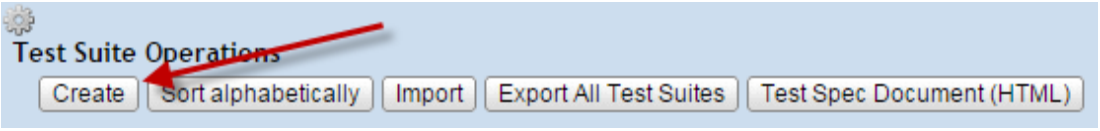


Illustration - 12 - Creating a test

7. Choose a suitable title for the test case. In the example the title "Registration – username with adjacent spaces" was chosen.
8. Optional: Write a short summary that describes the test purpose.
9. Click on the "Create" button to create the test.
10. The Test will now appear in the left navigation bar with an ongoing number and the chosen name. In this case it is: "FOE-1337: Registration – username with adjacent spaces."
11. When the test case is created in the FoE Automation project use the path and its name in the following style and put it in the field "TAP file name": com.innogames.qa.foe.tests.loginAndRegistration.FOE1336#registration_with_adjacent_spaces_in_username_FOE1337 . The "TAP file name" enables TestLink to access the test results.

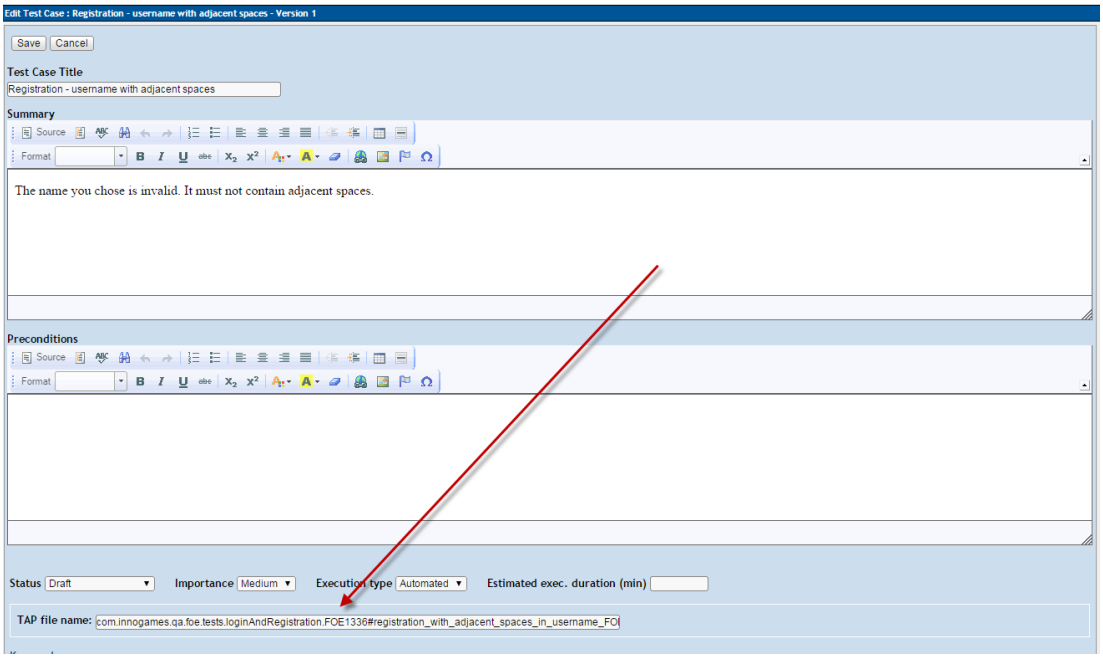


Illustration - 13 - Creating the "TAP file name"

12. Click on the small gear icon and then the "Add to Test Plans" button.
13. Do a checkmark at the "Test automation" Test Plan.

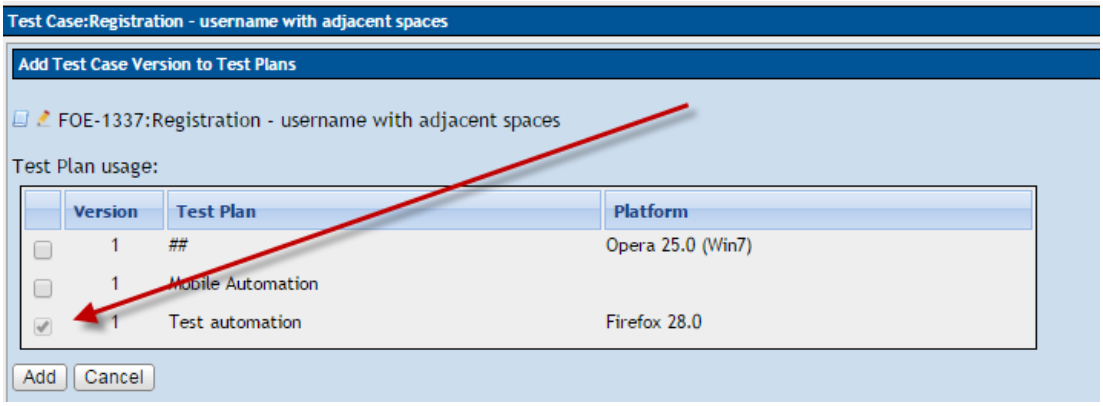


Illustration - 14 - Adding the test to the test plan

14. Click the "Add" button.

Once these steps are done it is time to create the test class in the FoE Automation project.

Creating a class in the FoE Automation Project

1. Create a class in the accordingly folder and name it like the ongoing number in TestLink. For the example it is "FOE1337" in "com.innogames.qa.foe.tests.loginAndRegistration"

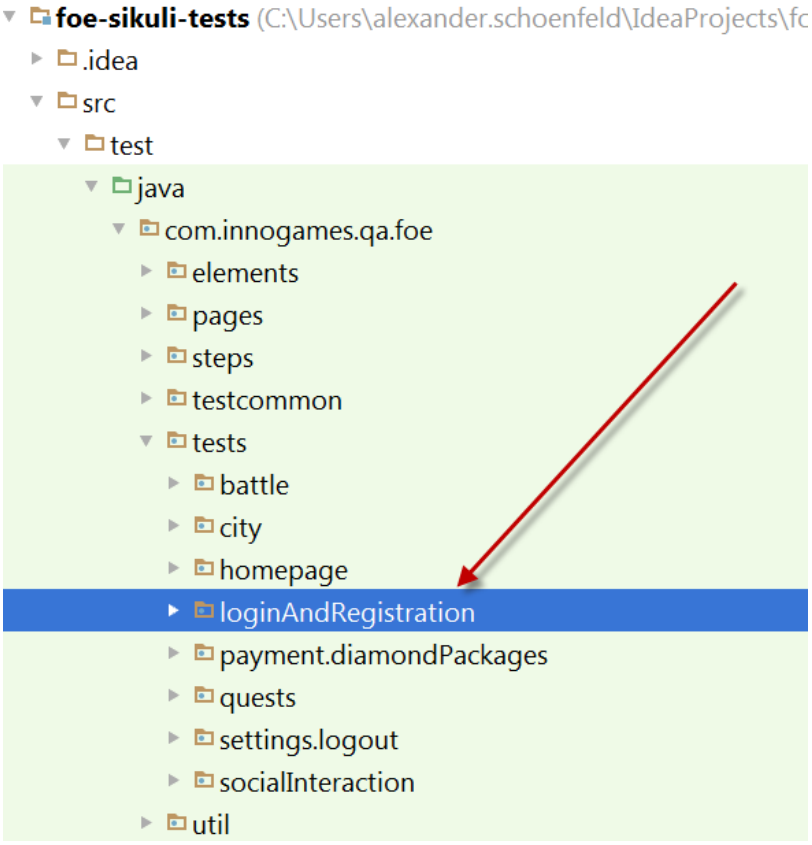


Illustration - 15 - Creating the test in the IDE

2. Extend the class with "SimpleTestTemplate"

3. Add the @WithTagValuesOf({“feature:login and registration”, “story: registration”, “level:basic”}) (Use fitting tags for every test. You can compare tags with related tags and copy them.
4. Within the class create a method with the @Test annotation on top and give it a comprehensible name with the ongoing number at the end. In this case its “registration_with_adjacent_spaces_in_username_FOE1337”.

```
@WithTagValuesOf({"feature:landing page", "story:registration",
"level:homepage", "top-level:smoke"})
public class FOE1337 extends SimpleTestTemplate {

@Test
public void registration_with_adjacent_spaces_in_username_foe1337()
throws IOException {
```

Code 19 – Creating the method

5. The first real step that needs to be done in the test is typing a user name into the register name field that has two adjacent spaces on the landing page. To define a step on the Landing Page the “LandingPage” PageObject needs to be created in the “pages” folder.
6. Extend that class with the “PageObject” class.
7. To access the register name field on the landing page create a WebElementFacade object and name it registerNameField.
8. Add the @Findby(xpath = "//input[@name='register_name']") annotation on top of the element to declare it. To find the xpath source go on http://yy.forgeofempires.com/ and do a right mouse click on the register name field and chose the “investigate element” option.

```
public class LandingPage extends PageObject {
    @FindBy(xpath = "//input[@name='register_name']")
    WebElementFacade registerNameField;
```

Code 20 – Creating *WebElementFacades*

9. Write a method that makes it possible to send input to that field and name it properly. Hand over a “String name” as a parameter and use it for the *“.sendKeys(name)”* method on the “registerNameField” object.

```
public void sendKeysToRegisterNameField(String name) {
        registerNameField.sendKeys(name);
}
```

Code 21 – Writing a method to type text in a field

10. Create a class named “AtLandingPage” in the “steps.guest” folder of the project.
11. Extend the “AtLandingPage” class with ScenarioStepsTemplate. This class contains all steps related to the landing page that can be reused in every test related to the landing page.
12. Create a “LandingPage” object and name it “onLandingPage”.
13. Create a method with the @Step annotation handing over a String “username” and access the “registerNameField” object through the

"onLandingPage" object using the ".sendKeysToRegisterNameField(username)" method.

```
public class AtLandingPage extends ScenarioStepsTemplate {

    LandingPage onLandingPage;
    RuntimePropertiesLoader runtimePropertiesLoader = new
RuntimePropertiesLoader();

@Step
public void logs_in_as(String username, String password)
throws IOException {
```

Code 22 – Creating a class to collect steps related to the Landing Page

14. Add a getter method in the "GuestSteps" class of the "steps" folder to make the "AtLandingPage" class accessible in the test classes.

```
public class GuestSteps extends ScenarioSteps {

    private StepFactory stepFactory;

    public GuestSteps(Pages pages) {
        super(pages);
        stepFactory = new StepFactory(pages);
    }

    public AtLandingPage atLandingPage() {
        return stepFactory.getStepLibraryFor(AtLandingPage.class);
    }
```

Code 23 – Adding a getter method to the GuestSteps class

15. Now everything is prepared to write the test easily in the test class.
16. Type "guest." and chose from the offered methods the ".atLandingPage()"
17. Chose the method that does the correct user interaction, which is in this case ".types_username_to_register("A B");"
18. Repeat steps 7-17 for the other steps. (Clicking the register button and checking for the correct error message)

```
@WithTagValuesOf({"feature:landing page", "story:registration",
"level:homepage", "top-level:smoke"})
public class FOE1337 extends SimpleTestTemplate {

@Test
public void registration_with_adjacent_spaces_in_username_foe1337()
throws IOException {
   guest.atLandingPage().types_username_to_register("A  B");
   guest.atLandingPage().clicks_register_button();
   guest.atLandingPage().should_see_error_message_invalid_username();
    }

}
```

Code 24 – Writing the test

Refactoring: From Spaghetticode to PageObject Pattern

This section describes the refactoring of the FoE Automation Project and writing new test cases in PageObject Pattern as part of a practical work of the thesis.

The term Spaghetticode [Spaghetti] is referring to code that has little structure which leads to a difficult maintainability and gives no opportunity for code reuse.

The illustration below shows a test case written in the above mentioned style. As one can see the code is not clear at first sight and it would take some time to understand the logic used. Though, the biggest disadvantage of that "style" is the non-existing possibility to reuse code, furthermore the missing ability for Thucydides to report detailed step information. Thucydides would only be able to report that an image has been clicked or that the mouse hovered over something. It would not be clear where the failure has occurred.

```
@WithTagValuesOf({"feature:continent map", "story:basic functionality", "level:battle"})
public class FOE5 extends CustomTestTemplate {

    @Test
    public void continent_map_is_shown_FOE5() throws IOException {

        logsSteps.atLogsSteps().log("Click on campaign map button and check if map is opened.");
        sikuliActions.clickOnImage(GameMainPage.CAMPAIGN_MAP_BUTTON);
        Assert.assertTrue("Continent map is not opened.",sikuliActions.isImagePresent(ContinentMap.PAGE_HEADER))

        logsSteps.atLogsSteps().log("Click on back ti city button and check is game menu bottom is visible.");
        sikuliActions.hoverMouseOver(GameMainPage.GLOBAL_CHAT_BUTTON);
        sikuliActions.clickOnImage(ContinentMap.BACK_TO_CITY_BUTTON);
        Assert.assertTrue("City view is not shown.",sikuliActions.isImagePresent(GameMainPage.GAME_MENU_BOTTOM))

    }
}
```

Illustration - 16 - „Spaghetti Code"

The great difference can be seen while comparing the Spaghetticode directly with the formatted PageObject Style. Almost like a normal text it is possible to read what actually happens. Step by step. Furthermore it is very efficient to repeat test steps through reusing the methods. This style of coding reduces the danger of careless mistakes and enables unexperienced programmers to write automated test cases.

```
@WithTagValuesOf({"feature:campaign map", "story:basic functionality", "level:battle", "top-level:smoke"})
public class FOE5 extends SimpleTestTemplate {

@Test
public void campaign_map_is_shown_FOE5() throws IOException {

guest.atLandingPage().logs_in_as_default_user(getClass().getSimpleName());
player.atLoginPage().clicks_play_button();
player.atWorldSelection().clicks_on_world_button(2);
player.atCityScreen().waits_for_town_hall_to_be_rendered(ScenarioStepsTempla
te.Eras.STONE_AGE);
player.atGameHud().clicks_campaign_map_button();
player.atGameHud().hovers_over_global_chat_button();
player.atCampaignContinentMap().clicks_back_to_city_button();
player.atGameHud().should_see_game_menu_bottom();}}
```

Code 25 - A test written with the Page Object design pattern

3. Conclusion

This thesis discussed general advantages and drawbacks of test automation in software projects and introduced an existing test automation framework implementation for an online strategy browser game. This test case has shown advantages as well as disadvantages of the introduced testing techniques. Those are discussed in the following. On the one hand there are the following benefits:

- Reuse of automated test cases: Once implemented the test cases can be repeatedly the same way without changing the conditions.
- Reproducibility of Errors: Often it is not possible to reproduce errors when the test cases are executed manually. With a good reporting system that serves as a good documentation test automation allows the reconstruction of errors.
- Regression testing: Once the code has changed, side effects can lead to errors in functionality that already has been tested. With manual testing it is often not possible to reiterate over all tests. Automated tests run much faster while causing nearly no additional human effort.
- Simulation of non-functional requirements: Performance tests must be automated in most cases, because manually, they are either too costly or impractical.
- Test coverage: Test automation can cover much more content in a shorter amount of time than manual testing can.

On the other hand there is a lot of decisions and planning that has to be done in the first place. In addition there are some drawbacks that should be taken into account:

- Test automation does not exclude human failure: The automated tests are still created by human beings so the possibility of failure is still given, even if it is just a false positive or wrongly evaluated test results.
- There is no guarantee that automated tests will find more bugs or that the total costs will be lower. Often the initial costs are underestimated and the only pay-off is only utilized after a long period.
- The creation of an automated test tool is difficult and requires skilled developers to set up the framework. In addition the creation of test cases has often to be done manually and the maintenance of them can be time consuming.
- The costs of refactoring the test framework and its tests can be underestimated, once the test software has changed.
- The evaluation of automated tests can lead to bottlenecks when there are differences in the target-performance comparison, because it causes a lot of manual evaluation.

All in all one can say that test automation is in most cases just reasonable in big or long-term software projects, but with a structured planning and a vision that maintains the motivation of the team it is possible to benefit from the advantages of test automation.

Applying these considerations to the FoE Automation project, one can say that the effort of developing this framework is worth it, even so the benefits are not yet measurable. Test automation in that sense has to be seen as a long term investment. The reason for this is the still low test coverage. Since FoE is a long term project it is very likely that the QA team will benefit from that investment of time and workforce. The reward will be a lot more time to focus on important features and test cases that are not possible to automate. Nevertheless the automation framework will need to be maintained and updated. This will require a lot of time when it comes to GUI changes, because the image database for the image recognition approach has to be updated. The decision to use the Page Object Pattern design was clearly a good one since it supports the reuse of code and simplifies the maintenance. In addition this comprehensible code will facilitate the access to software development for QA testers, who want to do further studies. Although Thucydides provides all tools and functionalities to turn the development process in ATDD, the FoE project does not utilize this opportunity. This, however, is already planned. Soon it will be possible to integrate the testing in the whole development process from the beginning. In the end one can say that state of the art browser games need test automation to stay maintainable.

4. List of Literature

Every web document is attached on the CD in the folder literature and is named like the abbreviation.

The date indicates the moment of data retrieval. The web links might differ from the web documents on the CD.

Abbr.	Web link	Date
[After]	JUnit API – Annotation Type After http://junit.sourceforge.net/javadoc/org/junit/After.html	08.01.2015
[AfterClass]	JUnit API – Annotation Type AfterClass http://junit.sourceforge.net/javadoc/org/junit/AfterClass.html	09.02.2015
[ATDD]	John Ferguson Smart – Acceptance test driven development for web applications http://www.javaworld.com/article/2078432/open-source-tools/acceptance-test-driven-development-for-web-applications.html	18.12.2014
[Before]	JUnit API – Annotation Type Before http://junit.sourceforge.net/javadoc/org/junit/Before.html	08.01.2015
[BeforeClass]	JUnit API – Annotation Type BeforeClass http://junit.sourceforge.net/javadoc/org/junit/BeforeClass.html	09.02.2015
[Browserstack]	Browserstack Homepage https://www.browserstack.com/automate	09.02.2015
[C&R]	Software and Programmer Efficiency Research Group – GUI Capture & Replay tools – http://sape.inf.usi.ch/gui-capture-replay-tools	01.02.2015
[ExImpl]	Selenium API – Explicit and Implicit Waits http://docs.seleniumhq.org/docs/04_webdriver_advanced.jsp	22.01.2014
[FindByAnnot]	Selenium Browser Automation Framework - @FindBy Annotation	22.01.2014
[InnoG]	Wikipedia – InnoGames http://en.wikipedia.org/wiki/InnoGames	09.01.2015
[Jenkins]	Wikipedia – Jenkins http://en.wikipedia.org/wiki/Jenkins_%28software%29	23.01.2015
[KDT]	Test Automation Patterns – Keyword Driven Testing http://www.dorothygraham.co.uk/patterns/desPatterns/keywordDriven.html	01.02.2015
[MBT]	AUTOMATED TEST CODE GENERATION FROM CLASS STATE MODELS - DIANXIANG XU, WEIFENG XU	11.01.2015
[Meth]	IX – Magazin für professionelle Informationstechnik – Ausgabe Dezember 2014 – Artikel: Unter Dauerstrom – S.52 – ISSN: 0935-9680	30.01.2015
[PageOb]	Selenium HQ – Page Object Pattern Design http://www.seleniumhq.org/docs/06_test_design_considerations.jsp#page-object-design-pattern	01.02.2015
[PGOP]	Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study -	14.01.2015

	Maurizio Leotta, Diego Clerissi, Filippo Ricca, Cristiano Spadaro	
[POP]	Assert Selenium - PageObject Pattern http://assertselenium.com/automation-design-practices/page-object-pattern/	16.12.2014
[RegTest]	Wikipedia – Regression Test http://de.wikipedia.org/wiki/Regressionstest	20.01.2015
[Sel]	Wikipedia - Selenium http://en.wikipedia.org/wiki/Selenium_%28software%29	15.12.2014
[Sik]	Sikuli – How Sikuli works http://sikulix-2014.readthedocs.org/en/latest/devs/system-design.html	19.12.2014
[Spaghetti]	Wikipedia – Spaghetti code http://en.wikipedia.org/wiki/Spaghetti_code	23.01.2015
[TAE]	A Theoretical Review of the Impact of Test Automation on Test Effectiveness - Donovan Lindsay Mulder and Grafton Whyte	01.01.2015
[Test]	JUnit API – Annotation Type Test http://junit.sourceforge.net/javadoc/org/junit/Test.html	08.01.2015
[Testautm]	Bitkom – Industrielle Softwareentwicklung – 4.3 Testautomatisierung http://www.bitkom.org/files/documents/Industrielle_Softwareentwicklung_web.pdf	20.01.2015
[TestLink]	Wikipedia – TestLink http://en.wikipedia.org/wiki/TestLink	08.01.2015
[Thucyd]	Thucydides Homepage http://www.thucydides.info/	15.12.2014
[ThucydPlgnJen]	Thucydides Plugin Jenkins https://wiki.jenkins-ci.org/display/JENKINS/Thucydides+Plugin	17.12.2014
[ThucydAnnot]	Thucydides Manual – Annotations http://thucydides.info/docs/thucydides-one-page/thucydides.html#_defining_high_level_tests_in_junit	15.12.2014
[ThucydHLRep]	Thucydides Manual - Defining high-level tests http://thucydides.info/docs/thucydides/_defining_high_level_tests_in_easyb.html	17.12.2014
[ThucydIntro]	Thucydides Manual - Introduction http://www.wakaleo.com/thucydides-one-page/thucydides.html#introduction	15.12.2014
[ThucydReport]	Thucydides Manual - Tags http://thucydides.info/docs/thucydides/_adding_tags_to_test_cases.html	16.12.2014
[ThucydRun]	Thucydides Manual – Thucydidesrunner.class http://wakaleo.com/thucydides-javadoc/net/thucydides/junit/runners/ThucydidesRunner.html	22.01.2014
[UserStory]	Wikipedia – User story http://en.wikipedia.org/wiki/User_story	17.12.2014
[W3C]	WebDriver Working Draft - Abstract http://www.w3.org/TR/2013/WD-webdriver-20130117/	15.12.2014
[WebDr]	WebDriver Homepage http://docs.seleniumhq.org/projects/webdriver/	15.12.2014
[WebDrAn]	Selenium API – Annotation Type FindBy https://selenium.googlecode.com/git/docs/api/java/org/ope	16.12.2014

	nqa/selenium/support/FindBy.html	
[WebDrITF]	Selenium API – WebDriver https://selenium.googlecode.com/svn/trunk/docs/api/java/org/openqa/selenium/WebDriver.html	15.12.2014
[WebEl]	Selenium API – WebElement https://selenium.googlecode.com/svn/trunk/docs/api/java/org/openqa/selenium/WebElement.html	15.12.2014

5. Eidesstattliche Erklärung

Ich versichere hiermit, die vorgelegte Arbeit in dem gemeldeten Zeitraum ohne fremde Hilfe verfasst und mich keiner anderen als der angegebenen Hilfsmittel und Quellen bedient zu haben.

Hamburg, den 16.02.2015

Unterschrift

(Vorname, Nachname)